Theoretical Computer Science 71 (1990) 209-226
North-Holland

MIXED COMPUTATION AND COMPILATION: NEW APPROACHES TO OLD PROBLEMS

Mikhail A. BULYONKOV

Computer Center, Siberian Division of the Academy of Sciences, Novosibirsk, USSR, 630090

Abstract. Mixed computation and partial evaluation are powerful programming tools which may be used for different program manipulations such as macroprocessing, compilation, debugging, installation and so on. The most interesting and most significant advances in the area of mixed computation and partial evaluation are connected with automatic compiler generation and were successful recently in Denmark, Japan, USSR and some other places. This paper presents a survey of different approaches to mixed computation and their comparison. It is shown that a non-trivial partial evaluator needs to use some developed mechanisms of polyvariancy and abstract interpretation. The importance and necessity of a user control over mixed computation is discussed. Some other kinds of program manipulations which can be realised via mixed computation are also considered. These are metaprogramming, incremental computation and sequential decomposition.

1. Introduction

First of all the very term, mixed computation, must be clarified. This term became quite stable in programming lexicon, but appears in various contexts with various meanings. The reason is partly in the definition itself; indeed, in most general cases mixed computations are defined [13] as an arbitrary mapping of the form $M: P \times D \to P \times D$, where P and D are the sets of programs and data, respectively, and M preserves some invariant Sem: if $M(p, d) = (p', d')$, then $\mathrm{Sem}(p, d) = \mathrm{Sem}(p', d')$. The preserved invariant has to agree with the standard semantics of the language but is not required to coincide with it.

Since there are no restrictions on the form or on the way of obtaining the result and any component of both argument and result may be set null, it is obvious that any conceivable program processor satisfies this definition. Nevertheless, such a general notion is undoubtedly useful as a methodological basis that helps to stress jointness and indissolubility of program and data transformation and aims to search for more general approaches to constructing program processors.

The term "mixed computation" is also used in a more restricted sense as a synonym to such notions as *specialization* [40], *projecting* [3] and *partial evaluation* [21].

Without going into details, projecting can be defined as adapting a program to a (partially known) external environment in order to obtain a more efficient program [22], i.e. compared to mixed computation in the broad sense, projecting fixes the results form (only programs) and the goal in general. Such treatment of projecting makes it practically indistinguishable with optimization with the *external environment*. Therefore one can regard projecting as a special kind of optimization.

Definition 1.1. Let p be some program and env be a predicate on the set of data. A program p_{env} is called a *projection on a domain defined by predicate* env or simply a *projection of p on* env if

$$\forall d \quad \text{env}(d) \Rightarrow p(d) = p_{env}(d).$$

The main requirement that distinguishes projection from other optimizations is its obligatory non-triviality. The fact is that actualization of mixed computation problems was caused by the discovery of fundamental relations between the main processes of compilation and interpretation.

Definition 1.2. A processor proj(P, Env) is called *a projector* of programs in a language L if proj(p, env) = p_{env}. If a projector is programmed in the same language L it is called an *autoprojector.*

Now let int(P, D) be an interpreter of a language LL, programmed in L and $p(D)$ some program in LL; so

$$\forall d \quad \text{int}(p, d) = p(d).$$

Then

$$\text{obj} = \text{proj}(\text{int}, (P = p)), \tag{F1}$$

$$\text{comp}(P) = \text{proj}(\text{proj}, (P = \text{int})), \tag{F2}$$

$$\text{cogen}(I) = \text{proj}(\text{proj}, (P = \text{proj})), \tag{F3}$$

where cogen(I) is a compiler generator that, given a program of interpreter int, generates a compiler comp from LL to L; obj = comp(p) is an object code of a program p. These relations are called Futamura projections after their first discoverer [21, 14].

Thus the autoprojector serves both as a tool and as a source material for automatic transformation of the language semantics given in the form of an interpreter into the compiler of this language. But with the theory of compilation being one of the most developed in computer science, a projector implementor not only can but also constantly has to check his solution, comparing it with the given one. (However, it should be remembered, that application of mixed computation to compilation is the main but not the only goal, and serves as "a touchstone".)

Thus, the main problem of mixed computation is non-triviality. From the purely mathematical point of view there is no interest in projecting, because on the functional level, the existence of the projection follows from the proof of Kleene's S^n_m theorem [29]. In the simplest case, projecting reduces to insertion at the beginning of the program of the so-called *explicators* of the form $X := x$, where X are the names of a part of the program arguments and x are their initial values.

However, even here not everything is that simple. To be a datum of a projector, an environment has to be defined in some language, which can be quite different

from the one in which the program is written. This brings another point of view on projecting [25, 30]: it is sufficient to transform the environment into the program that filters out data not satisfying this environment, and after that the problem can be solved by composition of obtained program with the source program.

Definition 1.3. A program [env] is called an *explicator* of the environment env if for any data d, d'

$$[\text{env}](d) = d' \Rightarrow \text{env}(d')$$

$$\text{env}(d) \Rightarrow [\text{env}](d) = d.$$

An explicator does not necessarily have the form of an assignment: if we ignore the possible incompatibility between the program language and the environment language the explicator [env] can be expressed as a conditional as in Fig. 1.

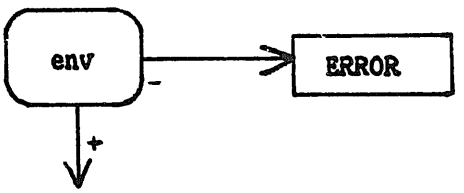


Fig. 1.

Definition 1.4. A program $p_1 \circ p_2$ is called a *composition* of programs p_1 and p_2 if

$$\forall d \quad p_1 \circ p_2(d) = p_2(p_1(d)).$$

Proposition 1.5. *For any program p and environment* env $p_{\text{env}} = [\text{env}] \circ p$.

Now let us get back to Futamura projections. It is obvious that if one takes the projector itself as interpreter, F3 becomes the special case of F2. Therefore a compiler generator, cogen, is a compiler for semantics defined by the projector proj. Now taking into consideration that semantics defined by a compiler and by an interpreter are equivalent (as denotational and operational semantics) we have that projection can be originally defined by the cogen processor. The following proposition holds.

Proposition 1.6. *Let* $g = \text{cogen}(p)$; *then* $g(\text{env}) = p_{\text{env}}$.

Proof.

$$g(\text{env}) = \text{proj}(\text{proj}, (P = p))(\text{env}) = \text{proj}_{(P=p)}(\text{env}) = \text{proj}(p, \text{env}) = p_{\text{env}}.$$

Verbally, the compiler generator generates a program, which in turn generates the residual program according to the environment. It turns out that any compiler generator realizes a projecting of some kind. Of course, in this case we have only an instrument, since we restrict the use of the projector by constructing all its possible projections by

$$\text{proj}(\text{proj}, (P = p)) = \text{cogen}(p). \qquad \square$$

The possibility of the double-phased implementation was discovered as an independent method and was called a *generating extension* method [15]. A program g was obtained as a result of substitution of some statements of the source program p by the output of their text. Now it is clear that these insertions are actually the reduced fragments of the projector and the generating extension itself is an object code for semantics defined by the projector.

To make the picture complete we shall cite one more point of view on projecting which led to a term *supercompilation* [44] (a prefix *super-* emphasis a meta-level of processing). The idea is to consider projecting as a generalization of a usual interpretation where all possible paths of execution are traced and unexecutable instructions are put aside in the residual program. However, in exactly the same way, one can consider a projector as a special kind of compiler with strongly developed constant propagation.

2. From strict scheme to polyvariant one

In spite of the variety of approaches to projecting and a lot of analogies with well-studied methods of program processing, it took a long time to pass from the declaration of the idea of obtaining compilers via mixed computation to its realization in a non-trivial projector. From a touchstone, Futamura projections turned into a real stumbling block. The first to announce, at the end of 1984, successful implementation of all three projections was Jones' group from DIKU [26, 42]. Almost at the same time, similar experiments were done at the Novosibirsk Computing Center [6] and Computing Center of Latvian University [4]. It is significant that these experiments covered both applicative and imperative programming languages. Soon there followed reports about similar projects for the languages based on term rewriting [39] and for Prolog [20].

For better understanding of the difficulties arising on the way, let us see how the first mixed evaluators worked [16]. As far as for linear fragments of the programs, everything was clear: all computations depending only on the accessible memory were carried out and all other computations were reduced and put aside in the residual program. The problem arose at the moment when the control reached a conditional whose condition was uncomputable using the accessible information. Indeed, at the meeting point of two different branches (called *focus* according to [16]) the states and even the sizes of the accessible memory could be quite different. So all computations inducing these distinctions had to be postponed. Information at the focus was made rough enough for the safest state. But even this turned out to be enough for this scheme of mixed computations (later called *strict*) to practically coincide with the trivial one at the first attempt to obtain an object code from the interpreter.

Ostrovsky proposed to relax the degree of this roughening by doing it not a priori but after carrying out computations on all branches and deleting only actually

different parts of accessible information [37]. This scheme was called *polyvariant mixed computation* [24]. But it was not much help in solving the above problem either. Roughly speaking, the interpreter represents a single loop with the body which looks through instruction types of the interpreted language, determines a needed alternative and chooses a successor. Then a typical fragment of the interpreter that processes the conditional looks like

```
... {evaluate the condition, put result to R0}
if R0 then
  cur_instr := then_successor[cur_instr]
else
  cur_instr := else_successor[cur_instr].
```

The value of R0 in general depends on the suspended data and therefore is inaccessible. It means that the variable cur_instr is suspended and after it, so does the whole loop of the interpreter.

A decisive step was taken when the polyvariancy principle was carried out in full measure. It required to restrict consideration to a class of programs, such that projecting would give a definitely non-trivial result by introducing a strict partition of the memory onto accessible and suspended parts [7]. Here only the accessible memory changes were traced and no roughening was admitted at all; when the branches with different states met, their continuations were processed independently (Romanenko even suggested the use of the terms *complete* and *disjointed* computations in contrast to *partial* and *mixed* ones [39]).

At once a question arose: "How to stop an avalanche-like multiplication of computation copies?" The solution is based on the fact that in the class of programs we are interested in (called in [7] *analyzer programs*), the set of accessible memory states is always finite. So observing the states appearing during computation and detecting repetition (which is inevitable on account of finite definiteness!) one can turn back the appropriate branch to the point of the residual program already produced. Just that very mechanism is a fundamental feature distinguishing the strictly polyvariant scheme.

The amount of information observed appears frightening at first sight but is in fact not so large. Indeed, taking the interpreter for an example we can see that during the computation, the program does not change, so it is only a relatively small totality of pointers to the program fragments and their attributes that actually control projecting. Besides, the appearing accessible memory states are not associated with every instruction of the projected program but only with the essential ones. Note that the real compilers perform the analogous actions filling the symbol and label tables.

The strictly polyvariant scheme eliminated the main defect of the previous schemes of projecting, i.e., the fact that the structure of the residual program was strictly

defined by the structure of the source program. Indeed, all structure alterations were implied by reductions of conditionals, since reduction of statements and loop unfolding changed only the length of the linear fragments. Now the residual program is formed as a folding of a potentially infinite tree of the source program expansion [8] and data structure (in our case the structure of the interpreted program) became the pattern for the projection structure.

It is interesting to note that the strictly polyvariant scheme of projecting first appeared under the name of "the most deep algorithm of partial evaluation" in Futamura's pioneering work [21], but it took him 11 years to rediscover his algorithm along with formulating its termination condition and realizing its consistency [22].

3. From partially defined memory states to environment

Somehow the projecting non-termination syndrome was overcome and almost at once computer experiments with autoprojectors were carried out. Thus the research turned from a demonstration of the possibility of mixed computation applications in principle, to proof and automatic implementation of specific mechanisms of compilation [9]. The problems that previously seemed technical turned into ones of principle and demanded a theoretical substantiation. One such problem is that it became insufficient to consider partially defined memory states without structured data. The latter are an essential part of any interpreter since its main argument, the program, is in fact a structured datum. It turned out that the data structure definiteness is somewhat orthogonal to the definiteness of the elementary components. Let us demonstrate it with examples. It may happen that specialization uses knowledge about data structure while the values of the components (or some part of the components) remains unknown. A typical example [36] is the list of pairs that link the names of the interpreted program with some values, for example (("x", ?), ("N", ?), ...). Here the list length is known and so is the first component of each pair; the second components are unknown. (Similar data representation was proposed in [17], where variable values were stored on the leaves of the parsing tree of a program. But the construction of the residual program there was carried out at demonstrational level only and did not contain technical details.) On the next metalevel, obtaining the compiler, the set of primary data is enlarged by the accessibility values [see also 39], e.g.

((KNOWN, UNKNOWN), (KNOWN, UNKNOWN), ...).

It is interesting that there exists a practical example when the component values are known but the structure is unknown [10]. Such is the control stack of the interpreter where the addresses of the program (evidently known) fragment are stored. But the order in which they are pushed on the stack and the depth of the stack depend upon suspended program data. Moreover the stack depth may turn out to be infinite, for example in the case of non-termination in recursive procedures.

The radical solution that gives a theoretical base to the above examples considers environment as an arbitrary predicate over memory states [8, 23, 32]. (Apparently the first analysis of environments of the more general form than the memory partial definiteness was done in [34] where they were called *areals*.) Thus in the base of mixed execution there is generalization of program constructions' semantics from memory state transformers to transformations of the memory states predicates, i.e. an *abstract interpretation*. That in particular explains fundamentality of polyvariancy as a reflection of predicate processing non-determinacy. The nature of polyvariancy is of two kinds: structure polyvariancy is caused by the ambiguous choice of successor in the conditional points; informational polyvariancy appears because of ambiguity of generalized predicate transformers.

A set of generalized protocols of the form

$$(p_1, \mathrm{env}_1), (p_2, \mathrm{env}_2) \ldots,$$

where p_i is some point in a program and env_i is an environment predicate, becomes a mixed computation invariant and the problem reduces to folding of this potentially infinite set of potentially infinite sequences into a finite object. In the extreme case, the source of finiteness is either the set of the appearing environments (a strict polyvariancy) or the finiteness of the source program that leads to the global data-flow analysis problem (e.g. [28, 41, 11]).

In the latter case as well as in all intermediate cases we need a sum operator + that approximates predicates disjunction.

Definition 3.1. Associative and commutative operation

$$+ : \mathrm{Env} \times \mathrm{Env} \to \mathrm{Env}$$

realizes a *sum* of environments if for any two environments env_1 and env_2 and data d

$$(\mathrm{env}_1 \vee \mathrm{env}_2)(d) \Rightarrow (\mathrm{env}_1 + \mathrm{env}_2)(d).$$

This definition is illustrated in Fig. 2.

The need for approximation is caused by the (already mentioned) fact that the arbitrariness of predicates is restricted by some language of environment definition. Operators of that kind were studied by many authors who named it *intersection* [24], *overlay* [37], *generalization* [45], *summing* [12]. But apparently we should take

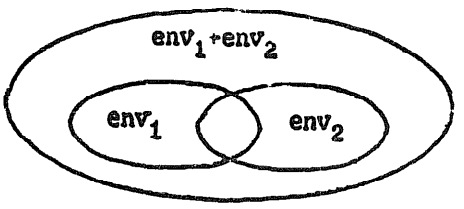


Fig. 2.

as a basis, an operation of assertion intersection in the global analysis problem, where a significant variety of assertion types is accumulated and the sufficient condition for termination is given: the set of assertions (in our terminology, the set of environments) are to be finite height semilattice for the order defined as

$$env_1 \leqslant env_2 \Leftrightarrow env_1 + env_2 = env_2.$$

Thus the information control (of which the strict polyvariant scheme is an example) may be defined by finer strategy than simple testing of environment equality.

Definition 3.2. Let C be some functional over set of environments Env

$$C : \text{Env} \to \text{Env},$$

that for each environment extracts some part of it

$$env \leqslant C(env).$$

The polyvariant projecting scheme is said to be *controlled by* C if two environments are declared compatible when the values of C on them are equal and also $env = C(env_1) = C(env_2) = C(env_1 + env_2)$ (see Fig. 3).

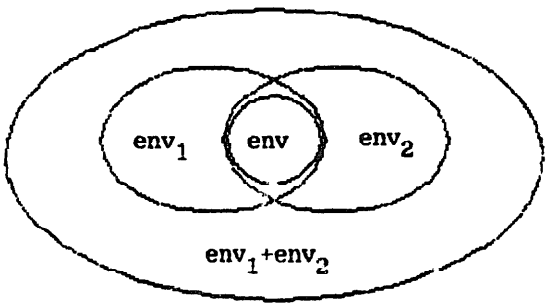


Fig. 3.

This approach is a kind of composition of strict and polyvariant schemes, being in its own way a step back from strict polyvariancy. This is shown to be a principal step, because the strict polyvariant scheme ignores the structure of the interpreter but good object code has to inherit and merge both the program's and the interpreter's properties.

4. On the implementation language

Now let us consider another important aspect of mixed computation application to compilation—the implementation language, i.e. the language in which auto-projector and interpreters of input languages are written. In [32] Lavrov gives the following reasoning.

Let the two arguments of the projector proj being defined in the languages L_1 and L_2, respectively. Since in the second Futamura projection

$$\text{comp}(P) = \text{proj}(\text{proj}, (P = \text{int})), \tag{F2}$$

the proj itself is the first argument, L_1 coincides with the implementation language L. The first projection

$$\text{obj} = \text{proj}(\text{int}, (P = p)) \tag{F1}$$

implies that $\text{int} \in L_1$. But in (F2) int is the second argument, so $L_1 = L_2 = L$. Thus the resulting object code appears to be the same language as the source program and the interpreter has to be an autointerpreter of the implementation language.

Here the error is concealed on the assumption that the implication

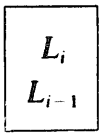
$$x \in S_1 \Rightarrow x \in S_2$$

in turn implies that $S_1 = S_2$ (where for x, S_1, S_2 there stand int, L_1, L_2, respectively). But actually L_2 is the data description language that envelops both the implementation language $L = L_1$ and all other input languages. For example we can choose as L_2 a description language of lists, or the set of all possible character sequences, etc.

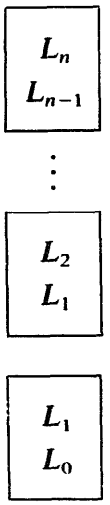
A problem of the implementation language level is more difficult and fundamental. On one hand it is the semantics description language, because the interpreters are being programmed in it, and its level must be sufficiently high. But on the other hand an object code is written in the same language and this fact supposes its closeness to machine code. It means that an implementation language has to contain both higher and lower level constructs: developed mechanisms of specification and composition along with computer oriented (or computer orientable) facilities; and it is a task of the interpreter designer to ensure that in the process of obtaining the object code all too high-level constructs would be "brushed away" by the projector. In other words, an implementation language must be a kind of high-level assembler. Exactly this tendency is now observed in the development of system programming languages. I wish to formulate the following.

Thesis 4.1. *A system programming language efficiency has to follow from its fitness for mixed execution.*

Another approach to a problem of the language's level uses the methods of metaprogramming, where the higher-order language concludes a chain $L_0 L_1 L_2 \ldots L_n$, where each L_i's semantics ($i \geq 1$) is defined by the interpreter written in terms of L_{i-1} and denoted as



Thus a "tower of interpreters" is constructed



Proposition 4.2

$$\text{proj}\left(\begin{bmatrix} L_1 \\ L_0 \end{bmatrix}, \begin{bmatrix} L_2 \\ L_1 \end{bmatrix}\right) = \begin{bmatrix} L_2 \\ L_0 \end{bmatrix}.$$

Proof. The projection of the first interpreter belongs to the same language L_0. The functional behavior of this projection can be derived as follows. For any program p in L_2, and any data d

$$\begin{bmatrix} L_1 \\ L_0 \end{bmatrix} \begin{bmatrix} L_2 \\ L_1 \end{bmatrix} (p\,d) = \begin{bmatrix} L_1 \\ L_0 \end{bmatrix} \left(\begin{bmatrix} L_2 \\ L_1 \end{bmatrix}, (p, d) \right) = \begin{bmatrix} L_2 \\ L_1 \end{bmatrix} (p, d) = p(d).$$

The first equality follows from the definition of projector and the second and the third follow from the definition of interpreter. So the projection under consideration realizes semantics of L_2. □

Such merging can be continued iteratively up to any level. Moreover, interpreters of all the languages written in L_0 and constructed this way may serve as a building material for all compilers to L_0 (of course only if a projector is an autoprojector). Note that the existence of a projector is in a certain sense a necessary condition for an efficient implementation of metaprogramming, since otherwise one has to define each language by compiling it into the previous one. And even then, instead of the inefficiency of the tower of interpreters, there would be an inefficiency due to the chain of compilers. Moreover, it would make disputable the advantages of metaprogramming; the main advantage is considered to be simplicity.

An interesting case of metaprogramming consists of the assumption that

$$L_0 \subseteq L_1 \subseteq L_2 \subseteq \cdots \subseteq L_n,$$

i.e. every next language is an extension of the previous one. So projecting can be used as a tool for implementing a special case of *boot-strapping*. But instead of a chain of compilers, we use here a chain of interpreters. The consistency of boot-strapping requires that the efficiency of the implementation of any language in the chain should not decrease with sequential reimplementation. Let us consider the sublanguage of L_{i+1} isomorphic to L_i and the corresponding subinterpreter which appears to be a self-interpreter. Then the following should hold

$$\text{proj}\left(\begin{bmatrix} L_i \\ L_0 \end{bmatrix}, \begin{bmatrix} L_i \\ L_i \end{bmatrix}\right) = \begin{bmatrix} L_i \\ L_i \end{bmatrix}.$$

If we instantiate i by zero then the following thesis can be formulated.

Thesis 4.3. *An efficient autoprojector for some language L should regenerate a self-interpreter of L by projecting it onto itself.*

This way of using mixed computation was first considered in [5] where the functioning of a program mixture was analysed; its different components were defined in the different level languages, or in the authors terms the languages of different intensity: the base language (L_0) was called "black" and the higher the lighter. Unfortunately this highly productive idea stayed unrealized and was later rediscovered, implemented and developed for the Prolog language [43].

Using Prolog as an implementation language also leads to better understanding of an incremental computation method which is related to metaprogramming and goes back to Lombardi's work [33]. The main point of the method is that the data are supplied by small portions gradually refining the program as formally stated in the following proposition.

Proposition 4.4. *Let p be a program and an environment* env *be represented as a conjunction of environments*

$$\text{env} = \text{env}_1 \,\&\, \text{env}_2 \,\&\, \cdots \,\&\, \text{env}_n,$$

then the projection p_{env} *can be obtained by the iterative process*

$$p^0 = p, \qquad p^{i+1} = p^i_{\text{env}_i} = \text{proj}(p^i, \text{env}_i).$$

Proof. The proof is by induction based on the rather evident fact that $p_{\text{env}_1} = (p_{\text{env}_1})_{\text{env}_2}$. □

This sort of execution can be found in the language INCOL [1, 2], whose semantics coincides with mixed computation. INCOL serves as a good example of efficiency and viability of a strict scheme of mixed computation since it was developed for a special class of programs, namely for economical tasks.

5. Automatic versus controlled mixed computation

Let us consider now whether a projector should be totally automatic. There are two points of view here. The first is developed in Ostrovsky's work [38] where the concept of controlled mixed computation is introduced. Its main point is that for the sake of efficient and profound projecting, the user is given a system of notions, inner constructs, accesses to a projected program, transformation instructions and also the means of composing them into different processors of mixed computation. These means can be various: a choice of alternative transformations (e.g. loop reduction or loop unfolding) made a priori or defined by the additional applicability conditions; using the metalanguage in which transformations are elementary instructions [18, 35]; etc. In fact here we deal not with a projector but with a projector construction set. The openness of such a system has a number of merits. First of all it is possible to take into account the specific character of a particular class of programs and to adapt a projector accordingly. Specific optimizations, auxiliary transformations (*conversions* according to [19]) and macro instructions supplementing standard types of control library are introduced in a natural way. A multiphase projecting becomes possible, that splits the process of obtaining the residual program into several conceptually closed primitive stages; each of them either uses a certain part of accessible information or prepares the program for the next stages. However, total flexibility of controlled mixed computation is also a disadvantage. The given opportunities presuppose the user's knowledge of hidden details which to a certain extent close the controlled mixed computations onto its implementor. Thus if a projector cannot handle a program, it is the projector that is being altered (exactly like the compiler designer searching for an error not in the program but in the compiler because the latter is closer to him). In the degenerate case for each program a special projector is created!

Another point of view declared by Jones' group [27] is that the (auto)projector has to be totally automatic. Moreover even the generated programs, in particular generating extensions and compilers, are alienated from the user, because in many respects they inherit inner constructs of a projector, unknown to him; that is, the user knows what the projector is doing but he does not and cannot know how it is doing this and therefore he has to adapt his program to the given projector. Thus in the extreme case the following thesis is realized.

Thesis 5.1. *A projector should process not any program but only a program specially developed for that one purpose.*

For example, consider a fragment of an interpreter implementing multiplication. If only the efficiency of the interpreter is taken into account the best way is simply to multiply the operands. But if the interpreter is intended for obtaining an object code (or a compiler) from it, then all cases should be considered: the left (the right) operand is equal to zero; the left (the right) operand is equal to one; the left (the

right) operand is a power of two, etc. We first have to be concerned about the object code efficiency. In general the quality of the residual program is compounded of the power of the projector and the suitability of the source program, where the influence of the latter can be an order of magnitude.

However, it turned out that there are also serious problems with this method. Many of them are considered in [27] and the main point is that not depending upon the projector's intellectuality, in some cases the information about the input data accessibility is not enough and it must receive additional information from the user about *how* to process this or that construct. Let us give two typical examples. Let the interpreter contain a special variable count_instr that keeps account of the number of executed instructions. Then despite the evident fact that reevaluation of this variable explicitly depends only upon the variable itself and not upon any inaccessible information,

count_instr := count_instr + 1,

it must be declared suspended in order to prevent the set of accessible memory states becoming infinite. Another example concerns one of the traditionally difficult questions of compilation—an adequate error handling. Thus, from the strict polyvariant scheme's point of view, the interpreter statements

print("Missing 'end'") and print("Division by zero")

are indistinguishable since it processes all program branches in the same way. But common sense suggests that the first one should be executed at the projection (compilation) stage whereas the second one should be carried over into the residual program (object code). So the user has to instruct the projector that in the latter case a more strict scheme has to be used, because the execution of this statement is determined, in general, by inaccessible data.

Prototypes of such instructions can be found in many studies. Danish autoprojector, for example, processes a strongly annotated source program: on the basis of the input data accessibility the first phase called a *bounding time analysis* automatically divides all parameters and function calls into static and dynamic ones. In the INCOL language there is a built-in function that determines whether a variable is accessible at a certain moment of execution (however, the usage of this function leads to incorrectness in mixed computation [46]). Here also go compile time variables, conditional compilation, all kinds of compiler directives and other similar mechanisms more or less characteristic of today's system programming languages.

It is clear that annotations controlling the projection are inserted only at some key points and the projector is to propagate them through the whole program. But the main question is the combination of automatic and controlled mixed computation at the expense of input language expansion by the projecting control tools. In an ideal case any construction can be related to either the first or the second stages of program execution, which actually embeds the projecting control language in the input language itself. Thus not only conditional compilation but also compile time

procedures, loops, structured variables input/output, etc. seem to be quite natural and relevant.

We can now have a new look at compilation itself. Instead of simple transformation of input language constructs into output language constructs, compilation covers the part of program execution that is possible due to operational semantics and input data accessibility. The boundary separating interpretation from object code execution becomes mobile and what is more, controllable. In this connection, compilation based on the patterns of code generation lies somewhere in the middle of the spectrum.

6. On optimizing compilation

Let us now briefly discuss optimizing compilation. First of all the object language optimization opt_0 and the source language optimization opt_1 must be distinguished (Fig. 4). The first is language-independent and can be formulated only once along with a projector and designed as a separate phase. The second has to be defined for each input language. Such optimization can be partly expressed in the form of special interpreter's conditions, which look for optimization patterns, i.e. the interpreter selectivity mentioned above. Another possibility is to give, along with the interpreter, some axiom system in order to define *an optimizing operational semantics* of the input language, but this is the problem for further investigation.

Thus, in general, optimizing compilation needs some special techniques differing from mixed computation. But nevertheless even if a non-optimizing compiler for a modest language is as simple as an interpreter for that language, projection allows one to avoid the problem of correspondence of different definitions of one object—the input language semantics.

7. From projecting to mixed computation

The above considerations are concerned only with projecting—mixed computation in the narrow sense. To show a significant difference between projecting and mixed computation in the broad sense let us consider the problem of generation of a sequence of compiler phases. Projecting allows one to extract a phase of compiling and execution. But since the result of projecting is only a program, then all projection derivates will produce only programs in the implementation language. But we wish to transform a program as a data into some other data being an *intermediate representation.* In addition this intermediate representation should be given the same meaning as that of the source program given by the language semantics. Thus besides an intermediate representation there must appear *intermediate interpreter* as suggested in [31].

Definition 7.1. *Partial evaluator* is a processor

$$\text{part}: P \times D \to P \times D$$

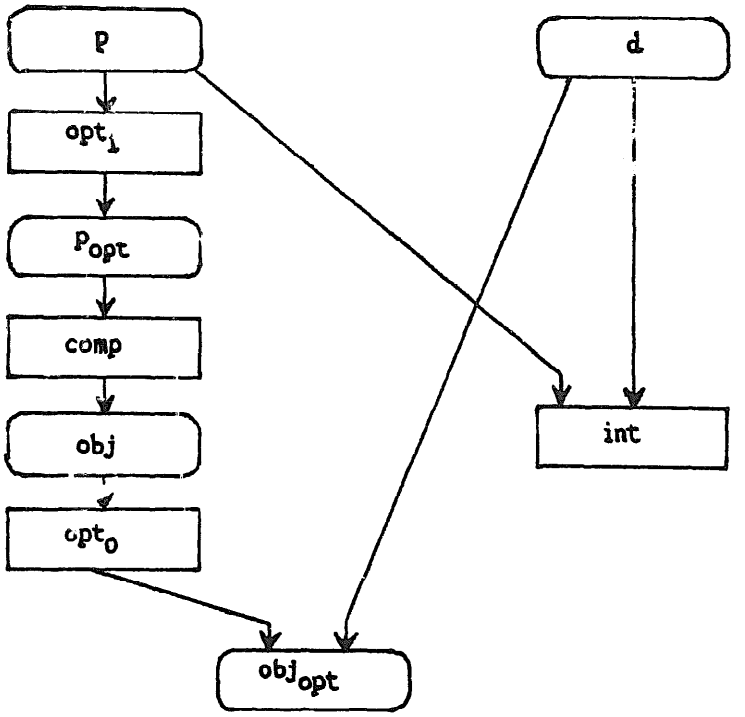


Fig. 4.

such that for any program p of two arguments and some data d_1 from

$$\text{part}(p, d_1) = (p', d'),$$

it follows that for any d_2

$$p(d_1, d_2) = p'(d', d_2).$$

Here p, d_1, p', and d_2 can stand for an input language interpreter, a source program, an intermediate interpreter, an intermediate program representation and data, respectively. The form of a processor part evidently makes us turn back to mixed computation in the broad sense. But it turns out that separation of compiling phases requires a mixed computation of a special kind [3], i.e. again the general notion has to be restricted but now in another direction: the form of the residual program must not depend on the accessible data. Actually it means that during construction of the residual program, the reductions are made (e.g. deleting instances of accessible memory names), and unfoldings, if being made at all, are made in a quantity that does not depend on particular values but is rather determined by the general characteristics of accessible memory. As for the typical inclusion of literal values evaluated on the accessible memory state in the residual program, they are screened by the fixed names of the intermediate memory.

The methods of obtaining the intermediate representation are sufficiently researched only in the case of unstructured memory [24]. But as we have already mentioned, application of mixed computation to compilation requires non-trivial processing of compound values. Drawing an analogy between partial evaluation of structured data and projecting of programs (which are also structured data) we can formulate two main questions:

(1) How is data structure reduced? For example how is an interpreter's array storing the processing program data split into scalar cells and sub-arrays?

(2) How is the data structure not present in the source program generated, for example, an attributed parsing tree, a transition graph, etc?

The first question has been systematically studied for lists [36] but requires further investigation for arbitrary data types. An approach to the solution of the second question was presented in [3]. Its basic idea is that the intermediate representation should be constructed in the form of an attributed graph with accessible memory states as nodes, the values of expressions dependent only upon the accessible memory as attributes and the program instructions, reevaluating the accessible memory states as the arcs. In the residual program all reevaluations of the accessible memory are transformed into instructions of intermediate representation graph traversal and expressions over the accessible memory—into addressing to the current node attributes values.

A question arises: how do the intermediate representations appear to be different? Really, a partial evaluator part disposes of the whole interpreter and the whole program and if there is no additional information then the intermediate representation has to be unique. Apparently, some of the interpreter instructions have to be declared suspended in spite of the fact that all their arguments are initially accessible. Step by step suspension relaxation in the iterative application of partial evaluation generates a sequence of intermediate representations. Here an analogy with incremental computation seems to be relevant, but now instead of progressive data entry, step by step interpreter instruction defreezing takes place.

The question of a mixed computation scheme lying between the extreme cases considered above still remains open. One of the most interesting subquestions here is which mechanisms and constructs are generated when such schemes are applied to compilation problems.

8. Conclusion

We have given a brief account of the state of art in applying mixed computation to compilation. It seems that in successful non-trivial experiments with mixed evaluators, the polyvariant projecting scheme and environment generalization for the structured data case were crucial. Nevertheless the great majority of these implementations are of an experimental or model nature. Evidently many present difficulties require fundamental theoretical study. Interacting development of the theory of mixed computation and the theory of compilation has to become fruitful for both, and even intermediate results of this process have to find practical applications.

Acknowledgment

I thank Neil D. Jones for fruitful discussion of the subject area and for very useful remarks on the paper during his visit to the USSR.

References

[1] G.Kh. Babich, L.F. Sternberg and T.I. Yuoganova, The INCOL programming language for incomplete information computation, *Programmirovanie* **4** (1976) 24–32 (in Russian).

[2] G.Kh. Babich et al., The INCOL programming language for incomplete information computation, *Upravlyayushchie Sistemy i Mashiny* (*Control Systems and Machines*) **4** (1982) 97–101 (in Russian).

[3] G.Ja. Barzdin and M.A. Bulyonkov, Mixed computation and compilation: Linearisation and decomposition of a compiler, Preprint 791, Computing Center of the Siberian Division of the Academy of Sciences, Novosibirsk, 1988.

[4] G. Barzdin, Experiments with mixed computation, *Programmirovanie* **1** (1987) 30–44 (in Russian).

[5] A.A. Bers, V.G. Polyakov and S.B. Rudnev, On a high-level programming system with mixed computation for personal microprocessor complexes, in: *Actual Problems of the Development of Architecture and Software of Computers and Systems* (Novosibirsk, 1983) 79–94.

[6] M.A. Bulyonkov, Computer experiment with mixed computation autoprojector, in: *Proc. 3rd All-Union Conference "Automation of the Production of Programming systems"*, Tallin, Polytechnical Institute of the Estonian Academy of Science (1986) 11–13 (in Russian).

[7] M.A. Bulyonkov, Polyvariant mixed computation for analyzer programs, *Acta Inform.* **21** (1984) 473–484.

[8] M.A. Bulyonkov, A theoretical approach to polyvariant computation, in: A.P. Ershov, D. Björner and N.D. Jones, eds., *Proc. Workshop on Partial and Mixed Computation*, GI. Avernaes, Denmark, October 1987 (North-Holland, Amsterdam, 1988) 51–64.

[9] M.A. Bulyonkov and A.P. Ershov, How do ad hoc compiler constructs appear in universal mixed computation process? in: A.P. Ershov, D. Björner and N.D. Jones, eds., *Proc. Workshop on Partial and Mixed Computation*, GI. Avernaes, Denmark, October 1987 (North-Holland, Amsterdam, 1988) 65–82.

[10] M.A. Bulyonkov, On obtaining an object code from one-loop interpreter, in: *Mathematical Theory of Programming* (Computing Center of the Siberian Division of the Academy of Sciences, Novosibirsk, 1985) 158–168 (in Russian).

[11] M.A. Bulyonkov, Iterative marking algorithms in transformational machine, in: *Software of Information* (Computing Center of the Siberian Division of the Academy of Sciences, Novosibirsk, 1982) 38–52 (in Russian).

[12] M.A. Bulyonkov, Mixed computation transformations, in: A.P. Ershov, ed., *Software Tools* (Computing Center of the Siberian Division of the Academy of Sciences, Novosibirsk, 1988).

[13] A.P. Ershov, On mixed computation: informal account of the strict and polyvariant computation schemes, in: M. Broy, ed., *Control Flow and Data Flow: Concepts of Distributed Programming* (Springer, Berlin, 1985) 107–120.

[14] A.P. Ershov, Mixed computation: potential application and problems for study, *Theoret. Comput. Sci.* **18** (1982) 41–67.

[15] A.P. Ershov, On the partial evaluation principle, *Inform. Process. Lett.* **6** (1977) 38–41.

[16] A.P. Ershov and V.E. Itkin, Correctness of the mixed computation in Algol-like programs, in: J. Gruska, ed., *Mathematical Foundation of Computer Science*, Lecture Notes in Computer Science **53** (Springer, Berlin, 1977) 59–77.

[17] A.P. Ershov, On essence of compilation, in: E.J. Neuhold, ed., *Formal Description of Programming Concepts* (North-Holland, Amsterdam, 1978) 391–420.

[18] A.P. Ershov, Transformational machine: theme and variations, in: *Lecture Notes in Computer Science* **118** (Springer, Berlin, 1981) 16–32.

[19] A.P. Ershov and B.N. Ostrovsky, Controlled mixed computation and its use to systematic development of language oriented parsers, in: *Proc. IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation*, Bad Tolz, FRG, 15–17 April (North-Holland, Amsterdam, 1987) 31–48.

[20] D.A. Fuller and A. Abramsky, Mixed computation of Prolog programs, *New Generation Comput.* **6**(2, 3) (1988) (to appear).

[21] Y. Futamura, Partial evaluation of computation process—an approach to a compiler-compiler, *Systems Computers Controls* **2** (1971) 45–50.

[22] Y. Futamura, Partial evaluation of programs, in: E. Goto et al., eds., *RIMS Symp. Software Science and Engineering, Kyoto 1982 Proceedings*, Lecture Notes in Computer Sciences **147** (Springer, Berlin, 1983) 1-35.

[23] Y. Futamura and K. Nogi, Generalised partial computation, in: D. Björner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988) 133-152.

[24] V.E. Itkin, On partial and mixed computation, in: A.P. Ershov, ed., *Program Optimization and Transformation, Part I* (Computing Center of the Siberian Division of the Academy of Sciences, Novosibirsk, 1983) 17-30 (in Russian).

[25] N.D. Jones and Mads Tofte, Some principles and notation of compiler generators, Unpublished working paper (1983).

[26] N.D. Jones, P. Sestoft and H. Sondergaard, An experiment in partial evaluation: the generation of a compiler generator, in: *Proc. 1st Int. Conf. on Rewriting Techniques and Application, Dijon, France, 1985*, Lecture Notes in Computer Science **202** (Springer, Berlin, 1985) 124-140.

[27] N.D. Jones, Automatic program specialization: A re-examination from basic principles, in: D. Björner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988) 225-282.

[28] J.B. Kam and J.D. Ullman, Monotone data flow analysis frameworks, *Acta Inform.* **7** (1977) 305-318.

[29] S.C. Kleene, *Introduction to Metamathematic* (Van Nostrand, New York, 1952).

[30] N.A. Krinitskii, An algorithmic analysis of mixed computation, *Programmirovanie* **3** (1987) 42-56 (in Russian).

[31] H. Kroger, A summary of a system for partial evaluation, residual evaluation, code generation and semantics directed compiler generation, Bericht Nr. 8505, Institut für Informatik und Praktische Mathematik, Universität Kiel, FRG, September 1985.

[32] S.S. Lavrov, On the essence of mixed computation, in: D. Björner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988) 317-324.

[33] L.A. Lombardi, Incremental computation, in: F.L. Alt and M. Rubinoff, eds., *Advances in Computers* **8** (Academic Press, New York, 1967) 247-333.

[34] M.S. Margolin, On a method of mixed computation implementation, in: *Automation of Application Packages Production* (Tallin, USSR, 1980) 111-113 (in Russian).

[35] A.P. Mel'nik, A program model of a transformational machine, in: *Problems of System and Theoretical Programming* (Novosibirsk, USSR, 1982) 25-34 (in Russian).

[36] T. Mogensen, Partially static structures in a self-applicable partial evaluator, in: D. Björner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988) 325-348.

[37] B.N. Ostrovsky, Application of mixed computation to systematic development of language-oriented parsers, in: *Compilation and Programs Models* (Novosibirsk, USSR, 1980) 69-80 (in Russian).

[38] B.N. Ostrovsky, Controlled mixed computation exemplified by language-oriented parsers, in: A.P. Ershov, ed., *Problems in Theoretical and System Programming* (Novosibirsk, USSR, 1984) 30-49.

[39] S.A. Romanenko, A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure, in: D. Björner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1983) 445-464.

[40] S.A. Romanenko, An application of mixed computation to assemblers and loaders, Preprint 27, Keldysh Institute of Applied Mathematics, Moscow, USSR, 1983 (in Russian).

[41] V.K. Sabelfeld, Equivalent transformations and program optimisation, Cand. Sci. thesis abstract, Novosibirsk (1980).

[42] P. Sestoft, The structure of a self-applicable partial evaluator, in: H. Ganzinger and N.D. Jones, eds., *Programs as Data Objects, Copenhagen, Denmark, 1985*, Lecture Notes in Computer Science **217** (Springer, Berlin, 1986) 236-256.

[43] A. Takeuchi and K. Furakawa, Partial evaluation of Prolog programs and its application to meta programming, in: H.-J. Kugler, ed., *Information Processing 86, Dublin, Ireland* (North-Holland, Amsterdam, 1986) 415-420.

[44] V.F. Turchin, A supercompiler system based on the language REFAL, *SIGPLAN Notices* **14** (1979) 46-54.

[45] V.F. Turchin, The algorithm of generalization in the supercompiler, in: D. Björner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988) 531-550.

[46] T.I. Youganova, Mixed computation correctness: INCOL language example, *Programmirovanie* **2** (1987) 67-77 (in Russian).